**Cluster Architecture**

* + **ETCD** is a database that stores information in a key value format.
  + **Schedulers** identifies the right node to place a container on based on the containers.
  + Controller take care of container health.
  + **Node controller** takes care of nodes. They are responsible for onboarding new node in the cluster.
  + **Replication controller** ensures that the desired number of containers are running at all the time in your application.
  + **Kube API** server is primary management component of Kubernetes. It is responsible for orchestrating all operations within the cluster. It exposes the Kubernetes API, which is used by external users to perform management operation on the cluster as well as the various controllers to monitor the state of the cluster and make necessary changes as required and by the worker node to communicate to the server.
  + A **Kubelet** is an agent that runs on each node in a cluster. It listen the instruction from the Kubernetes API server and deploys or destroy containers on the nodes as required. The Kube API server periodically fetches the status reports from the Kubelet to monitor the status of the nodes and container on them.
  + **Kube proxy** responsible for communication between the worker nodes. It ensures that the necessary rules are in place in the worker nodes to allow the containers running on them to reach each other.

Kubernetes Architecture

**ETCD for Beginners**

* What is ETCD?

It is a distributed reliable key-value store that is simple, secure and fast.

* What is a Key-Value Store?

It stores the information in the form of a documents or pages.

* How to get started quickly?

Install the ETCD

1. Download Binaries
2. Extract
3. Run ETCD Service “./etcd” use to run the service

* How to operate ETCD?

1. The ETCD is configured only in the master nodes
2. When we run, ETC service then it start on a port 2379 by default.
3. The default client that comes with ETCD is ETCD controller.
4. The ETCD control client is a command line client for ETCD
5. We can use it to store sand retrieve key value pairs.
6. To store a key value pair run the **“./etcdctl set kue1 value1”** command.
7. To retrieve the store data run the **“./etcdctl get key1**” command
8. To view more option run **“./etcdctl”** without argument.

**ETCD in Kubernetes**

* “--advertise-client-urls https://$(INTERNAL\_IP):2379” – the is the address on which ETCD listens
* “etcdctrl get / --prefix –key-only” – list all the key stored by coordinators.
* Kubernetes stores data in a specific directory structure.
* In a HA Environment, we will have multiple master node in your cluster then we will have multiple ETCD instances spread across the master nodes. In such case, make sure that the ETCD instances know about each other by setting the right parameter in the ETCD service configuration.

“initial-cluster controller-0=https://${CONTROLLER0\_IP}:2380, controller-1=https://${CONTROLLER0\_IP}:2380”

**Kube-API Server**

* + It is the primary management component In Kubernetes.
  + When we run “**kubectl**” command then the Kube-control utility is reaching to the Kube API server. Kube API server first authenticates the request, validated it, and then retrieves the date from the ETCD cluster and response back with the request of information.
  + We do not really need to use **kubectl** command line instead, we could invoke the API directly by sending the POST requests.
  + Kube-API server is responsible for authenticating and validating requests, retrieving and updating data in the ETCD data store.
  + Kube-API server is the only component that interacts directly with the ETCD data source.
  + The other component like scheduler, Kube-controller manager, Kubelet uses the API server to perform updates in the cluster respective areas.
* How to view the api-server options in an existing server?

#cat /etc/kubeernetes/manifests/kube-api-server.yml #yaml file

#cat /etc/system/system/kube-apiserver.service ##server location

**Controller managers**

* A controller is a process that continuously monitors the state of various component within the system and works towards bringing the whole system to the desired functioning state. For example:

1. Node controller is responsible for monitoring the status of the nodes and taking the necessary action to keep the application running. It does that through Kube-API server. The node controller takes the status of the nodes every 5 seconds. That way node controller can monitor the health of the nodes if it stop receiving heartbeat from a node the node is marked as unreachable but it waits 40 seconds before marking it unreachable after a node is marked unreachable it gives it five minutes to come back up if it does not, it removes pods and assigned to that node a and provisions them on the healthy one, if the pods are the part of a replica set.
2. The next controller is replication controller. It responsible for monitoring the status of replicas set and ensuring that the desired number of pods are available at all times within the set. If a pod dies, it creates another one.

* The entire controller package into a single process known as “Kubernetes controller manager”.
* How do you install and view the Kubernetes controller manager?

1. Download the Kube-controller manager from the Kubernetes release page.
2. Extract it
3. Run it as a service.

#kube-controller-managerservice

**Kube proxy**

* + Within a Kubernetes Cluster, Every pod can reach every other pod. This is accomplished by deploying a pod networking solution to the cluster.
  + A pod network is an internal virtual network that spans across all the nodes in the cluster to which all the paths connected to this network, they are able to communicate with each other.
  + There are many solutions available for deploying such a network.
  + Example:
    - A web application deployed in the first node and a database application deployed in the other node. The web application can reach out the database by using IP of the pod. But there is no guarantee that the IP of the database pod will always remain the same. A better way for the web application to access the database is using a service to recreate a service to expose the database application across the cluster. The web application can now access the database using the name of the service.
    - The service also gets an IP address assigned to it whenever a pod tries to reach the service using its IP and name, it forwards the traffic to the back end pod.
  + The service cannot join the pod network because the service is not an actual thing. It is not a container like pod so it does not have any interfaces or an actively listening process. It is a virtual component that only lives in the Kubernetes memory.
  + Kube proxy is a process that runs on each node in the kubernetes cluster.
  + Its job is to look for new service and every time a new service is created, it creates the rules on each node to forward traffic to those service to the backend pods.
  + One way, it does this is using IP tables rules. In this case, it creates an IP tables rule on each node in the cluster to forward traffic heading to the IP of the service which is 10.96.0.12 to the IP of the actual pod which is 10.32.0.15. so that’s how kube proxy configured as a service.
  + How to install kube-proxy?
    - Download the kube-proxy with the kubernetes release page.
    - Extract it.
    - Run it as a service
  + The “kubeadm” tool deploys kube-proxy as pod on each node. In fact, it is deployed as a demon set. So a single pod is always deployed on each node in the cluster.

**Pod**

* Kubernetes does not deploy container directly on the worker nodes. The containers are encapsulated into a Kubernetes as object knows as Pod.
* A pod is a single instance of an application.
* A pod is smallest object that you can create in Kubernetes.
* Pods usually have a one to one relationship with containers running your application.
* A single Pod can have multiple containers except for the fact that they are usually not multiple containers of the same kind.
* The two containers can communicate with each other directly by referring to each other as local host since they share the same network.
* The two containers can easily share the same storage space as well.
* Note: multiple containers are a rare use case.
* How to deploy pods?

#kubectl run ngnix –image ngnix

* List the pod available in the cluster

#kubectl get pod

**ReplicaSet (Replication Controller)**

* Controllers are the process that monitors Kubernetes object and respond accordingly.
* Replication controller helps us to run multiple instances of a single pod in the Kubernetes which providing the high availability.
* It ensures that the specified number of pods is running at all the time.
* It creates multiple pods to share the load across them
* There are two similar term Replication controller and Replica set. Both have the same purpose but they are not same.
* Replication controller is the older technology that is being replaced by Replica set. Replica set is the new recommended way to set up replication.
* How to create Replication controller?
  + - By using Yaml file, we will create replication controller
    - As per the definition of the Kubernetes file, there are 4 section:

1. apiVerion – it is specific to what we are creating. For example, in this case replication controller is supported in Kubernetes API version v1.

2. kind – what type of Kubernetes resource you want to create. For example in this case, we will use as “ReplicationController”

3.metadata – this is use to name the Kubernetes resource.

4.spec – This is very critical part of the definition. It will define what will be created. We will create “template” section under spec to provide a pod template to be used by the replication controller to create replicas.

* The Yaml file(rc-defination.uml) for replccas controller is:

apiVerion: v1

kind: ReplicationController

metadata:

name: myapp-rc

labels:

app: myapp

type: front-end

spec:

template:

metadata:

name: myapp-pod

labels:

type: front-end

spec:

containers:

- ngnix-container

Image: ngnix

replicas: 3

#kubectl create –f rc-defination.yml

* To view the list of the Replication controller

# kubectl get replicationcontroller

* How to create Replicaset?
* It is very similar to replication controller. As usual, we have apiversion, kind, metadata and spec. The apiVersion is bit different which is “apps/v1”. The   
  “kind” will be “replicaset”. “labels” and “name” in the metadata. “spec” section looks very similar to the Replication controller. It has a “template” section where we provide pod definition and it has “replicas” section where we provide the number of the pod.
* The Yaml file(rc-defination.uml) for replccas controller is:

apiVerion: apps/v1

kind: ReplicationController

metadata:

name: myapp-rc

labels:

app: myapp

type: front-end

spec:

template:

metadata:

name: myapp-pod

labels:

type: front-end

spec:

containers:

- ngnix-container

Image: ngnix

replicas: 3

selector:

matchLabels:

type: front-end

#kubectl create –f rc-defination.yml

* To view the list of the Replication controller

# kubectl get replicaset

* There is one major difference between Replication controller and Replica set. Replica set requires a **selector** definition. The selector helps the replica set to identify what pods fall under it. The selector is one of the major differences between replication controller and replica set. The selector is not a required field in case of a replication controller but it is available when you skip it. It assumes it to be the same as the labels provided in the pod definition file. In case of the Replica set, a user input is required for this property and it has to be written in the form of match labels.
* A “match labels” selector imply matches the labels specified under it to the labels on the pod.
* The Replica set also provide many other options for matching labels that were not available in the Replication controller.
* How to scale the replica set?
* There is multiple way to do it:

1. Update the number of “replica” in the definition file. Then run the below command that will update the replca set to the new number.

# kubectl replace –f replicaset-defination.yml

1. Run the “kubectl scale” command, use the “—replicas” parameter to provide the new number of replicas and specify the same file as input.

#kubectl scale –replicas=6 –f replicaset-defination.yml

1. We may input the definition file or provide the Replicas set name in the type name format.

#kubectl scale –replicas=6 replicaset myapp-replicaset

* Commands:
* Create the kuberentes resource based on the input mention in the file.

# kubectl create –f replicasset-defination.yml

* List the replica set in the Kubernetes cluster

# kubectl get replicaset

* Delete the Kubernetes resource based on the input mention in the file.

# kubectl delete –f replicasset-defination.yml

* To replace and update the replicas set

# kubectl replace –f replicaset-defination.yml

* To replace and update the replicas set

# kubectl scale –replicas=6 –f replicaset-defination.yml

**Deployment**

* Pods deploy single instances of our application such as web application. Each container is encapsulated in pod. Multiple such pods are deployed using Replication controller and Replica set and then comes Deployment which is a Kubernetes object that comes higher in the hierarchy.
* The deployment provides us with the capability to upgrade the underlying instances seamlessly using rolling updates under the changes and pods and resume changes as required.
* How to create Deployment?
* The Deployment definition file are exactly similar to the replicas set definition file except for the kind which is now going to be Deployment.
* Below is the deployment file

apiVerion: apps/v1

kind: Deployment

metadata:

name: myapp-rc

labels:

app: myapp

type: front-end

spec:

template:

metadata:

name: myapp-pod

labels:

type: front-end

spec:

containers:

- ngnix-container

Image: ngnix

replicas: 3

selector:

matchLabels:

type: front-end

#kubectl create –f rc-defination.yml

* To view the list of the Deployment controller

# kubectl get deployment

* To view the list of the Replication controller

# kubectl get replicaset

* Commands:
* To see all the created objects at once run the below command:

# kubectl get all

**Namespaces**

* When Kubernetes cluster creates then it create below name space:

1. Default namespace –
2. Kube-system namespace - Kubernetes creates a set of pods and services for its internal purpose such as those required by the Networking solution, The DNS service etc. to isolate these from the user and to prevent from accidentally deleting or modifying the services. Kubernetes creates them under this name space.
3. Kube-public – This is where resources that should be made available to all users are created.

* We can create our own name spaces as well.
* Each of the names spaces can have its own set of policies that define who can do what.
* We can also assign quota of resources to each of the name spaces that way each name space is guaranteed a certain amount and does not use more than its allowed limit.
* Resources within a namespace can refer to each other by name.

For example, the web pod in the default namespace to connect to the database in the dev environment or name space then use "**service\_name.namespce.svc.cluster.local**" format.

* Looking closely at the DNS name of the service, the last part “cluster.local” is the default domain in the Kubernetes cluster. “svc” is the sub domain for service followed by “namespace” and then “name of the service”.
* Operational Aspect of namespace

#kubectl get pod ## it will list all the pod in the default name spce

#kubectl get pods –namespace=kube-system ##list all the pod in the kube-system name space

#kubectl create –f pod.yml ##create a pod in the default name space

#kubectl create –f pod.yml –namespce=dev ##to create a pod in the dev namespace.

* How to create a NameSpace?
* Like other yml file

apiVersion: v1

kind: Namespace

metadata:

name: dev

spec:

hard:

pods: “10”

requests.cpu: “4”

requests.memory: “5Gi”

limit.cpu: “10”

limit.memory: 10Gi

* Run the below command to create the namespace

#kubectl create –f namespace.yml

* Another way to create name space

#kubectl create namespace <namespace-name>

* How to change the default name space permanently?

#kubectl config set-config $(kubectl config current-context) –namespce=<new-namespce>

* Contexts are used to manage multiple clusters in multiple environments from the same management system.
* TO limit a resources in a namespace create a resource quota.
* To view the pod in all namespaces

#kubectl get pod --all

**Service**

* Kubernetes services enable communication between various component within and outside of the application.
* Kubernetes services helps us connect application together with other applications or users.
* One of the use case of service is to listen to port on the node and forward request on that port to a port on the pod running the web application. This type of a service is known as node port service because the service listens to a port on the node and forward requests to the pod.
* Different type of service:

1. NodePort – the service makes an internal pod accessible on a port on the node.
2. ClusterIP – The service creates a virtual IP inside the cluster to enable commination between different services such as a set of front-end server to a set of back end servers.
3. Load balancer – We provisions a load balancer for our application in supported cloud provides.

* NodePort Kubernetes service
* There are 3 port involved in the NodePort

1. Target pod – The port on the pod where the actual web server is running.
2. Port – it is the port on the service itself.
3. Node port – we use to access the web server externally.

* NodePort can only be in a valid range, which is default from 30,000 to 32767.
* How to create a service?
* We will use the definition (Yaml) file to create a service.

apiVersion: v1

kind: Service

metadata:

name: myapp-service

spec:

type: NodePort

ports:

- targetPort: 80

Port: 80

NodePort: 30008

Selector:

App: mysapp

Type: front-end

* Note: Out of all these, the only is the mandatory field is “port”. If we don’t provide a “target port”, it is assumed to be the same as “port”. And if we don’t provide “NodePort” then a free port in the valid range between 30,000 to 32767 is automatically allocate. Also, note that port is an array. Therefore, we can have multiple such port mapping within a single service

#kubectl create –f service-defination.yml

#kubectl get services

**Services Cluster IP**

* Each service gets an IP and name assign to it inside the cluster and that is the name should be used by other pods to access the service. Such types of service is known as Cluster IP.
* How to create a service?

apiVersion: v1

kind: service

metadata:

name: back-end

spec:

type: ClusterIP

ports:

- targetPort: 80

Port: 80

Selector:

App: myapp

Type: back-end

#kubectl create –f service-defination.yml

**Scheduling**

**Manual Scheduling**

* How scheduler works in the backend?
* Every pod has a field called “nodeName” that by default is not set. We don’t tipically specify this field when we create the pod. Kubernetes add it automatically.

The scheduler goes through all the pods and look for those that do not have this property set. Those are the candidates for scheduling.

* It then identifies the right node for the pod by running the scheduling algorithm.

Once identified, it schedules the pod on the node by setting the “nodeName” property to the name of the node by creating a binding object.

* So if there is no scheduler to monitor and schedule nodes, the pods continue to be in a pending state. In such case, we manually assign pod to nodes.

Without a scheduler the easiest way to schedule a pod is to set the “nodeName” field to the name of the node in the pod specification file while creating the pod.

* If the pod is already created and you want to assign the pod to a node, Kubernetes won’t allow modifying the “nodeName” property of a pod.
* So another way to assign a node to an existing pod is to create a binding object and send a post request to the pod binding API then mimicking what the actual scheduler does in the binding object, we specify a target node with the name of the node then send post request to the pod binding API with the data set to the binding object in a Jason format. So we must convert Yml file into its equivalent Jason form.

**Labels and Selectors**

* It is the standard method to group things together.
* Labels are properties attached to each item.
* Selector helps us to filter the items.
* How to specify labels in Kubernetes?
* In the pod definition file, under “metadata” create a section called “labels”. Under that add the labels in the key value format .
* We can add as many as labels we like.
* Once a pod is created, to select the pod where the labels assign. Use “kubectl” command along with the selector option and specify the condition.

#kubectl get pod –selector app=App1

* Kubernetes object use labels and selectors internally to connect a replica set consisting of three different pods. We first label the pod definition and use selectors in a replicas set to group the pod in a replica set definition file. You will see labels definition in two places. The labels define under the template section are labels configured on the pod, the labels define under the top(metadata) are the labels of the replica set. The labels of the replica set will be used if we were to configure some other object to discover the replica set. in order to connect the replica set to the pod, we configure “selector” field under the replicas set specification to match the labels defined on the post. A single label will do if matches correctly.

**Annotations**

It is used to record details for inflammatory purpose. For example tool details like name version build information etc.

**Resource Requirement and limits**

* Kubernetes schedulers decide which node a pod goes to. The scheduler takes into consideration the amount of resource required by a pod and those available on the nodes. In this case the schedulers schedules a new pod on node if the node has no sufficient resource, then the schedulers avoid placing the part on that node instead places the pod where sufficient resource are available. If there is no sufficient resource available, Kubernetes holds back scheduling the pod. You will see the pod in a pending state. You can see the message in the events.
* By default Kubernetes assume that a pod or a container with in a pod requires 0.5 CPUs and 256 bytes of memory. This is known as resource request for a container. We can modify the default CPU and memory value by specifying them in the pod or deployment definition files. For example:

Spec:

Containers:

- name: simple-webapp-color

Image: simple-webapp-color

Ports:

* + - * containerPort: 8080

Resources:

Requests:

Momory: “1Gi”

CPU: 1

Limits:

Memory” “2Gi”

CPU: 2

* By default Kubernetes, sets a limit of 1 vCPU 2 containers.
* By default Kubernetes, sets limit of 512 bytes on containers.
* We can change the limit by specify in the definition file as above.
* Node: Limits and requests are set for each container within the pod.
* When the pod tries to exceed resource beyond its specified limit, in case of CPU, it throttles the CPU so that it does not go beyond the specified limit. A container cannot use more CPU resources than its limit. But this is not the case of the memory. The container can use more memory resource than its limit. So if a pod tries to consume more memory than its limit constantly, the pod will be terminated.

**Daemon sets**

* Daemon sets are like replica sets as it helps us to deploy multiple instances of pod but it runs one copy of the pod on each node in the cluster.
* Whenever a new node is added to the cluster a replica of the pod is automatically added to that node and when the node is removes then pod is automatically removed.
* The daemon set ensures that one copy of the pod is always present in all nodes in the cluster.
* Use cases of Daemon set:
* Deploy a monitoring agent or log collector on each of our nodes in the cluster.
* The Kube-proxy is one of the good use cases of demon set. The kube-proxy component can be deployed as a demon set in the cluster.
* Networking solutions like eve vive requires an agent to be deployed on each node in the cluster.
* How to create a Demon set?
* Creating a Daemon set is similar to the replica set creation process.

Below is the yaml file

apiVerion: v1

kind: DaemonSet

metadata:

name: monitoring-daemon

spec:

selector:

matchLabels:

app: monitoring-agent

template:

metadata:

labels:

spec:

containers:

- monitoring-agent

Image: monitoring-agent

#kubectl create –f daemon-set-defination.yml

* To view the Daemon set on the cluster

# kubectl get daemonsets

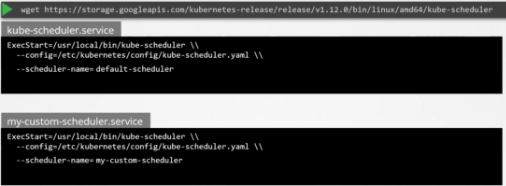
* To view the more details:

#kubectl describe daemonsets <daemo- set-name>

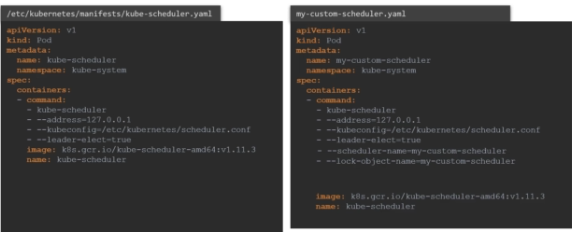
* How does Daemon set works?
* On each pod set the “nodeName” property in its specification before it is created and when they are created, they automatically land on the respective node. That’s how we use till Kubernetes 1.0.12 version. From version above 1.0.12, the Daemon set uses the default schedulers and Node affinity rule that we have learnt on one of the previous topic.

**Multiple Schedulers**

* We can define our own schedulers. SO while creating a pod or deployment, we can instruct Kubernetes to have the pod scheduled by a specific schedulers.
* How to deploy the Kube-shedulers?
* Download the binaries and run it as a service with a set of option. One of the option is “—sheduler-name” if not specified then it will take the default scheduler.
* To deploy the additional schedulers, we will use the same binaries and set the schedulers name ““—sheduler-name=my-custom-schedulers”



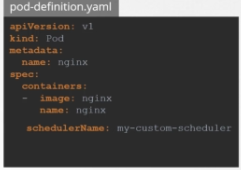
* How to deploy schedulers using Kubeadm tool?



* An important option to look here is “—leader-elect” option is used when we have multiple copies of the schedulers running on different master nodes. In HA stetup where we have multiple master nodes with the kube-schedulers running on both of them.
* If the multiple copies of the same schedulers are running on different nodes, only one can be active at a time. It will help in choosing the leader.
* To get multiple schedulers working, we must set the ““—leader-elect” option “false”, in case we don’t have the multiple master. If we have the multiple master then we can pass an additional parameter which is “—lock-object-name=my-custom-schedulers” to set a lock .
* To view the new custom schedulers

#kubectl get pods –namespace=kube-system

* Configure the new pod or deployment using new schedulers
* In the pod specification file add a new field called “schedulerName: my-custom-schedulers



* By viewing the event, we will get to know which schedulers the pod picked up.

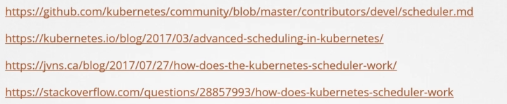
#kubectl get events

* To view the logs of the schedulers

#kubectl logs my-custom-schedulers –name-space=kube-system

**Configuration Kubernetes Schedulers**

Some of the advance topic in the Kubernetes:



Logging and Monitoring

**Monitoring Cluster Component**

* + As of now, Kubernetes does not come with a full feature built in monitoring solution. However there are number of open source solution available today such as metrics server Prometheus, elastic stack and propriety solution like data dog and diner trace.
  + Heapster was one of the original projects that enabled monitoring and analysis features for Kubernetes. Heapster is deprecated now and slimmed down version formed known as the Metric server.
  + We can have one metric server per Kubernetes cluster. The metric server retrieves metric from of the Kubernetes node and pods, aggregates them and stored them in memory.
  + Note that the metrics server is only an in memory monitoring solution and does not store the metrics on the desk and as a result you cannot see historical performance data.
  + Kubernetes runs an agent on each node known as “Kubelet” which is responsible for receiving instructions from the Kubernetes API master server and running pod on the nodes. The Kubelet also contains also contain a sub component know as C advisor or container adviser. The C- adviser is responsible for retrieving performance metrics from the pod and exposing them through Kubelet API to meet the metrics available for the metrics server.
  + Deploying the metric server
* Clone the repository

#git clone <https://github.com/kubernetes-incubator/metrics-server>

* Then deploy the service with the “kubectl” command

#kubectl create –f deploy/1.8\*/

* Once configured, then we can few the performace metrics with

#kubectl top node

* To view the performance the metrics of the pod

#kubectl top pod

**Managing Application Logs**

* When we create a pod in Kubernetes using pod definition file.

#kubectl create –f pod.yml

* Once the pod is running, we can view the logs using

#kubectl logs –f <pod-meta-data>

* If we have multiple container inside the pod, we can view the logs

#kubectl logs –f <pod-meta-data> <name-of-the-container>

**Application Lifecycle Management**

**Rolling updates and Rollbacks**

To see the status of the rollout

#kubectl rollout status deployment/myapp-deployment

To see the revisions and history of rollout

#kubectl rollout history deployment/myapp-deployment

Deployment Strategy – there are 2 types of the deployment strategies

1. Recreate strategy – According to this strategy, first destroy the previous version and then create the new version. In this strategy, the application will be down for some time and this is not the default strategy.
2. Roiling strategy– According this strategy, we do not destroy all the pod at once instead we take down the older version and bring up a newer version one by one. This way application never goes down and the upgrade is seamless. It is the default deployment strategy.

* Note: if we don’t specify a strategy while creating the deployment, it will assume it to be rolling update.
* To update the image of the application, we can use “kubectl set image” command

#kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1

* The difference between recreate and rolling update strategies can also be seen when we view the deployment in detail. Run the “kubectl describe deployment” command to see the details information regarding the deployment. In the recreate case, the old replica set was scale down to zero first and then the new replica set scaled up to fine and the in the rolling update strategy, the old replicas set was scale down one at a time and simultaneously scaling up the new replica set one at a time.
* Kubernetes deployment allow us to roll back to a previous revision to undo a change. Then the deployment will destroy the pod in the new replica set and bring the older ones up in the old replica set.

#kubectl rollout undo deployment/myapp-deployment

* Command
* Below command will create both pod and deployment.

#kubectl run ngnix –image=ngnix

* To apply and set image command to update the deployment

#kubectl apply –f deployment-defination.yml

#kubectl set image deployment/myapp-deployment nginx=ngnix:1.9.1

* To see the rollout status command

#kubectl rollout status deployment/myapp-deployment

* To rollback a deployment operation

#kubectl rollout undo deployment/myapp-deployment

* To see the rollout history

#kubectl rollout history deployment/myapp-deployment

**Commands**

* We will see the commands and argument in the Kubernetes.
* The “ENTRYPOINT” instruction is like the command instruction as in we can specify in the program that will be run when container starts and whatever we specify on the command line will get appended to the entry point.
* In case of the CMD instruction, the command line parameters passed will get replace entirely whereas in case of entry point the command like parameter will get appended.
* We can use both ENTRYPOINT and CMD in the same docker file when one was not specified in the command line. In such case the command instruction will be appended to the entry point instruction. We should always specify the ENTRYPOINT and CMD instruction in the json format.

**Commands and Argument**

* We will create a pod using the image which will sleep for 5 sec

apiVersion: v1

Kind: Pod

Metadata:

Name: Ubuntu-sleeper-pod

Spec:

Containers:

- name: Ubuntu-sleeper

Image: Ubuntu-sleeper

Command: [“sleep”]

Args: [“10”]

* If we want need the container to sleep for 10 seconds, then we can specify the additional argument in the pod definition file like above.
* Anything that is appended to the “docker run” command will go in to the “args” property of that pod definition in the form of an array like above.
* The ENTRYPOINT is the command that is run at startup and CMD is the default parameter to the command with the args option in the pod definition file.
* The command field overrides the ENTRYPOINT instruction and the args field overrides the command instruction in the docke file.
* ‘ENTRYPOINT[“sleep”]’ in docker file is equal to ‘command’ field in the pod definition.
* CMD[“5”] in docker file is equal to ‘args[“10”]’field in the pod definition.

**Configuring Environment Variables in Application**

* Given in a pod definition file which uses the same image as the docker.
* To set an environment variable, we use “env” property.
* “env” is an array. So every item under the “env” property starts with a dash(-), indicating an item in the array. Each item has a name and a value property.

env:

- name: APP\_COLOR

value: pink

* There are other ways of setting the environment variable such as using config maps and secrets. The difference is that instead of specifying value we see “valuefrom” and then a specification of config map or secret.

env

- name: APP\_COLOR

valuefrom:

configMapKeyRef:

env

- name: APP\_COLOR

valuefrom:

secretKeyRef:

**Configuring ConfigMaps in Application**

* ConfigMaps are used to pass configuration data in the form of key value pairs in Kubernetes.
* When a pod is created, then we inject the config map into pod. So the key value pairs that are available as environment variables for the application hosted inside the container in the pod.
* There are 2 phases involved in configuring config map:

1. Create the config map
2. Inject them into a pod

* There are two ways to create the config map:

1. The imperative way without using a config map definition file

#kubectl create configmap <config-name> --from-literal=<key>=<value>

#kubectl create configmap app-config --from-literal=APP\_COLOR=blue

# kubectl create configmap <config-name> --from-file=<path-to file>

# kubectl create configmap app-config --from-file=app\_config.properties

1. The declaratory by using a config map definition file.

apiVersion: v1

kind: ConfigMap

metadata:

name: app-config

data:

APP\_COLOR: blue

APP\_MODE: pod

#kubectl create –f configMap.yml

To view the configMap

#kubectl get configmaps

To describe the configMap command

#kubectl describe comfigmaps

* Configuring the configMap with Pod

apiVersion: v1

kind: ConfigMap

metadata:

spec:

containers:

- name: simple-webapp-color

Image: simple-webapp-color

Ports:

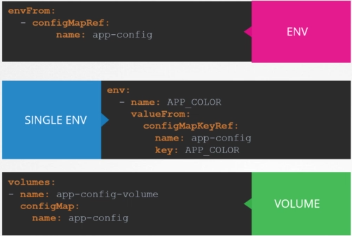
- containerPort: 8080

envFrom:

- configMapRef:

Name: app-config

* “EnvForm” property is a list so we can pass as many environment variable as required.
* Each item in the list corresponds to a configmap item, specify the name of the config map we create earlier.
* There are others way to inject configuration data into pods:



**Configure Secrets in Applications**

* ConfigMap stores the configuration date in plain text format.
* Secrets are used to stroe sensitive information like passwords or key
* Secrets are similar to config maps except that the secret stored in an encoded or hashed format
* There are 2 steps involved in working with secrets

1. Create the secret
2. Inject them into pod

* There are 2 ways of creating secret:

1. Imperative way without using s select definition file

#kubectl create secret generic <secret-name> --from-literals=<key>=<value>

# kubectl create secret generic app-secret --from-literals=DB\_HOST=mysql

# kubectl create secret generic <secret-name> --from-literals=<path-to-file>

# kubectl create secret generic app-secret --from-file=app\_secret.properties

1. Declarative way by using a secret file

apiVersion: v1

kind: Secret

metadata:

name: app-secret

data:

DB\_HOST: mysql

DB\_User: root

DB\_Password: passwd

* While creating a secret with declarative approach, we must specify the secret values in hashed format.
* Run the below command to convert the value in the hash

#echo –n ‘mysql’ | base64

* To view the secret

#kubectl get secrets

* To describe the secret

#kubectl describe secrets

* To see the secret with full details in yml formate

#kubectl get secret app-secret –o yml

* We can decode the hash value with the below command

#echo –n ‘bxlzlwn’ | base64 –decode

* Configuring the secret with Pod

apiVersion:

kind:

metadata:

name: simple-webapp-color

labels:

name: simple-webapp-color

spec:

containers:

- name: simple-webapp-color

Image: simple-webapp-color

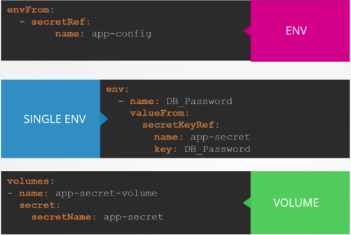
Ports:

- containerPort: 8080

envFrom:

- secretRef:

Name: app-secret



**Self Healing Applications**

* Kubernetes supports self-healing applications through ReplicaSets and Replication Controllers.
* The replication controller helps in ensuring that a POD is re-created automatically when the application within the POD crashes.
* It helps in ensuring enough replicas of the application are running at all times.

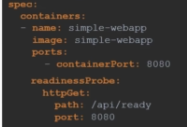
Readiness Probes

**Readiness Probes**

* A pod has a pod status and some condition. The pod status tells us where the pod in its lifecycle.
* When a pod first created, it is in the pending state. This is when thheschedulerstries tofigure out where to place the pod.
* If the schedulers cannot find the node toplace the pod, it remains in a pending state.
* To find out why it's stuck in a pending state run the “kubectl describe pod” and it will tell you exactly why.
* Once the pod is scheduled, it goes into a container creating status where the images required for the application are pulled and the container starts. Once all the containers in a pod start, it goes into a running state, where it continues to be until the program completes successfully or is terminated.
* Note - At any point in time, the pod status can only be one and only gives us a high level summary of a pod.
* “Pod conditions” compliment pod status. It is an array of true or false values that tell us the state of a pod when a pod is scheduled on a node, the pod scheduled condition is set to true when the pod is initialized.
* To see the status of the pod condition run the “kubectl describe pod” command and look for the condition section.
* There are different ways that we can define if an application inside a container is actually ready. We can set up different kinds of tests or probes. For example:

1. In case of web application, It could be even the API server is up and running. So you could run it as TTP test to see if the API server response.
2. In case of a database you may test if a particular TCP socket is listening or simply execute a command within the container to run a custom script that will exit successfully if the application is ready.

* How do we configure the test in the pod definition file?
* Add a new field called “readinessprobe” and use the” httpGet” option with specify the “port” and the ready API.
* Now when the container is created, Kubernetes is not immediately set the ready condition on the container to true.

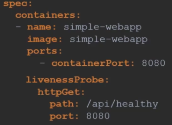


* There are different ways a probe can be configured.
* For a HTTP use the “ httpGet” option with the pod.
* For the TCP use the “ tcpSocket” option with the “port”.
* For executing a command, specify the “exec” option with the command add option in an array format.
* If we know that the application will take minimum of 10 sec to warm up, we can add an additional delay to the probe
* We can also specify how often to probe, then we can do that with the option “periodseconds” option
* By default if application is not ready then after 3 attempts, then probe will stop.
* We can specify more probe attempts by using “failureThreshold” option.



**lliveness Probe**

* A lliveness probe can be configured on the container to periodically test whether the application within the container is actusally healthy. If the test fails, the container is considered unhealthy and is destroyed and recreated
* The liveness Probe is configured in the pod definition definition file same as “readiness Probe”. Except Except here you use “livenessProbe” instead of “ readinessProbe” .



* Similar to readiness probe, we have “httpdGet” probe for API , TCP socket for ports and “exec” for commands as well as addition option like initial delay before the test is run, Period seconds to define the frequency and success and failure thresholds.



**Cluster Maintenance**

**OS upgrade**

* If a node is down for more than five minutes then the pods are terminated from that node. It is consider as dead.
* If a node is down for a 5min then Kubernetes master consider it dead and new pod will creates on the other node.
* So whenever downtime is required, tell the Kubernetes master that will be down. So that Kubernetes shift the entire load to the different node and release the server for maintenance.
* The time to waits for a pod to come back online is known as the pod eviction timeout and it is set on the controller manager with a default of five minutes.

#kube-controller-manager --pod-eviction-timeout=5m0s

* We can drain the nodes purposefully, so that the workload are moved t other nodes. So when we drain a node than the pods are gracefully terminated from the node and recreated on another node.

#kubectl drain node-1

* The node is also cordoned or marked as un-schedulable until we specifically remove the restriction. Once the node comes back then again we need to uncoordinated manually to work the node as excepted.

#kubectl uncordon node-1

* There is one more option called “cordon”. Unlike drain, It will simple mark the node un-schedulable. It does not terminate or move the part on an existing node. It simply makes sure that new parts are not scheduled.

**Kubernetes Software Versions**

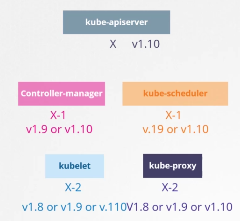
* To see the Kubernetes version across the cluster

#kubectl get nodes

* The Kubernetes released versions (v1.11.3) consist of three parts. The first is the major version followed by the minor version and then the patch version.
* Minor versions are released every few months with new features and functionalities.
* Patches are released more often with critical bug fixes.
* Kubernetes follows a standard software release versioning procedure every few months.
* The first major version 1.0 was released in July of 2015.
* The latest stable version is 1.0 13 as of now.
* All the bug fixes and improvements first go into an alpha release tagged alpha in this release. In this release, the features are disabled by default and maybe buggy. Then from there they make their way to beta release where the code is well tested. The new features are enabled by default. And finally they make their way to the main stable release
* Note - there are other components within the Kubernetes that do not have the same version number. The ETC cluster and core DNS servers have their own versions as they are separate projects.

**Cluster Upgrade Process**

* The components can be at different release versions since the kube API server is the primary component in the control plane and that is the component that all other components talked to
* None of the other components should ever be at a version higher than the Kube API server.
* The controller manager and scheduler can be at a one version lower than API server.
* Kubelet and kube-proxy component can be at two versions lower than API server
* This is not the case with “kubectl” utility. The “kubectl” can be higher than one version or equal to the version or one less version than the Kube-apiserver.



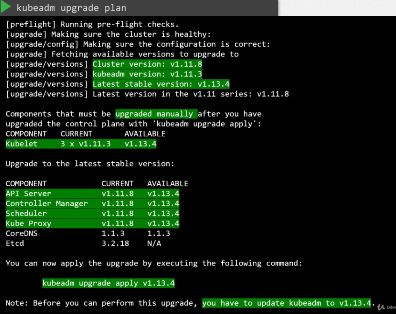
* At any time, Kubernetes support only up to the recent three minor versions.
* The recommended approach is to upgrade one minor version at a time.
* Below command can help us to upgrade the cluster

#kubectl upgrade plan

#kubectl upgrade apply

* Upgrading a cluster involves two major steps:

1. Upgrade the Master node - While the master is being upgraded, the control plane components such as the API server Scheduler and controller managers go down briefly. The master of going down does not mean the worker nodes and applications on the cluster are impacted. All workloads hosted on the worker nodes continue to serve users as normal. Since the Master is down, all management functions are down. We cannot access the cluster using ‘kubectl’ or Kubernetes API. Also we cannot deploy new applications or delete or modify existing ones. The controller managers don't function either. If a pod fail then a new pod will not create.
2. Upgrade the worker node - There are different strategies available to upgrade the worker nodes.
   1. One is to upgrade the entire worker node at once but then your ports are down and users are no longer able to access. This strategy that requires downtime.
   2. The second strategy is to upgrade one node as a time.
   3. A third strategy would be to add new nodes to the cluster nodes with newer software version. This is especially convenient if you're on a cloud environment where you can easily provisioned new nodes and decommission old ones.



‘

* Kubeadm” does not install or upgrade “kubelet”
* Upgrading the Kubernetes from 1.11 to 1.13
* We can only go once version more that the instlled one.
* So we will go 1.11 to 1.12
* First upgrade the “kubeadm” tool

#agt-get upgrade –y kubeadm=1.12.0-00

* Then upgrade the cluster

#kubeadm upgrade apply v1.12.0

* So now the cluster upgrade to 1.12
* If we will run the below command then we will still see the old version of Kubernetes. This is because it showing the version of the “kubelet” on each of the node register with API server not the version of the API server itself

#kubectl get node

* Upgrade the kubelets on the master node, if we haveinstalled.

#apt-get upgrade –y kubelet=1.12.0-00

#systemctl restart kubelet.

#kubectl get node

* “kubelet” version of worker node is still 1.11. So now upgrade the worker node one at a time. Move the workload from the first worker node to the other nodes.

#kubectl drain node-1

#apt-get upgrade –y kubeadm1.12.0-00

#apt-get upgrade –y kubelet=1.12.0-00

#kubeadm upgrade node –kubelet-version v1.12.0 ##update the node configuration for the new kubelet version

#systemctl restart kublet

#kubectl uncordon node-1

**Backup and Restore**

Things we should take backup in the Kubernetes:

Resource configuration

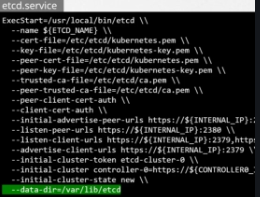
ETC cluster

Persistent Volumes

* Store the Resource configuration file in the github
* A better approach to backing up resource configuration is to query the “kube-api-server”. Query the “kube-api-server” using the “kubectl” or by accessing the API server directly and save all the rsource configuration for all objects is created on the cluster as a copy. For example:

#kubectl get –all-namespaces –o yaml > all-deploy-service.yaml

* Tools like ARC or VELERO is use to take the configuration back for Kubernetes.
* The ETC cluster stores information about the state of our cluster. So information about the cluster itself. The nodes and every other resource as created within the cluster are stored here.
* So instead of backing up resource as above we may choose to back up the ETCD server itself
* As we know ETCD server is hosted on the master node, So while configuring ETCD with specified a location where all the data would be stored.

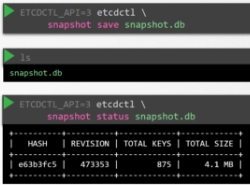


* ETCD also comes with a built in snapshot solution. We can take a snapshot of the ETCD database by using “etcdctl” utiity

#etcdctl snapsort save snapshot.db

* View the status of the Backup

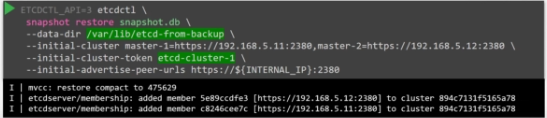
#etcdctl snapsort status snapshot.db



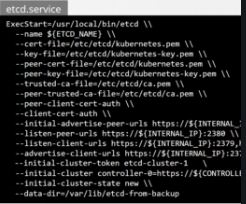
* To store the cluster from the above backup at any time
* Stop the kube-apiserver

#service kube-apiserver stop

* Run the below command

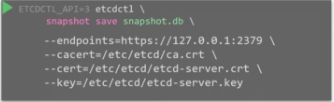


* When ETCD restore from the backup, it initialized as a new cluster configuration and configure the members ETCD as new members to a new cluster. This is to prevent a new member from accidentally joining an existing cluster. So during a restore we must specify a new cluster token and the same initial cluster configuration options specified in the original configuration file.
* We then configure the ETCD configuration file to use the new cluster token and data directory and then restart the ETCD service

****



* Note – With All the ETCD commands, Remember to specify the certificate files for authentication, Specify the end point to the ETCD cluster, CA certificate, ETCD server certificate and the key. Like below



* If we are using managed Kubernetes environment, service, then at a time, we do not have the access to the ETCD cluster. In such case, Backup by querying the kub-api server is probably the better way.

**Security**

**Kubernetes Security Primitives**

* Access of the host must be secured.

1. root access disabled
2. Password based authentication disable.
3. Only SSH Key based authentication to be made available.

* As we know, kube-api server is at the center of all operations within Kubernetes. We interact with it through “kubectl” utility or by accessing the API directly. Through that, we can perform almost any operation on the cluster. Therefore, the first thing we need to secure is controlling access to the API server.
* We need to make two types of decisions:

1. Who can access the cluster?

* Who can access the API server is defined by the authentication mechanisms.

There are different to authenticate to the API server

1. User I.D. and password stored in static files.
2. tokens
3. Certificates
4. Integration with external authentication providers like ldap.
5. Service accounts
6. What can they do?

* It is defined by authorization mechanisms.
* Authorization is implemented using role based access controls where users are associated to groups with specific permissions.
* In addition, there are other authorization modules like the attribute based access control, not authorizers, weblogs etc.
* All communication with the cluster between the various components such as ETCD cluster, the kube-controller manager, scheduler, API server as well those running on the worker nodes such as the Kubelet and kube-proxy, is secured using tier less encryption.
* By default, all pod can access all other pod within the cluster. We can restrict access between them using network policies.

**Authentication**

* Below person is accessing the Kubernetes cluster.

1. Admins
2. Developer
3. Application End user - Security of end users who access the applications deployed on the cluster is managed by the applications themselves internally. So we no need to worry on this.
4. Bots or third party –

* They are two types of user who is accessing the Kubernetes Cluster.

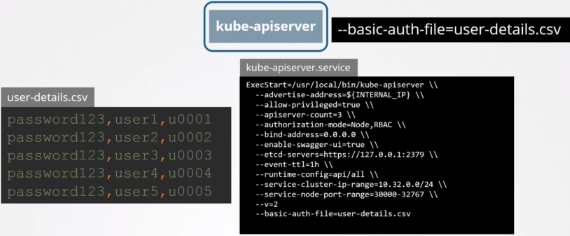
1. Humans such as Admins, Developers etc
2. Robert such as other process or applications that require access to the clusters.

* Kubernetes does not manages user accounts natively. It relies on an external source like a file with user details or certificates or a third party identity like LDAP to manage these users. So we cannot create users in the Kubernetes cluster or view the list of users.
* We can create and manage service accounts using Kubernetes API.

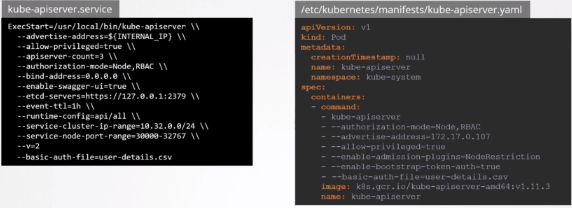
#kubectl create serviceaccount <user-name>

#kubectl list serviceaccount ##it will list all the service account.

* All user access is managed by the API server. The Kube-api server authenticates the requests before processing it.
* There are different authentication mechanisms.
* We can have a list of user name and passwords in a static password file or user names and tokens in a static token file or we can authenticate using certificates. And another option is to connect to third party authentication protocols like LDAP Kerbaros Rose etc.
* Authentication with Static password and token files
* We can create a list of users and their passwords in a CSP file and use that as the source for user. The file has three columns password, user name and user I.D.
* We then pass the file name as an option to the cube API server
* We must restart the Kubernetes server to make this change effective.



* If we set up your cluster using the kubeadm , then we must modify the kube-api server pod definition file. The kubeadm tool automatically restart the kube-api server once the file is update.



* To authenticate using the basic credentials while accessing the API server specify the user and password in a curl command.



* Similarly, like static password file, we can use the static token file to specify the token.



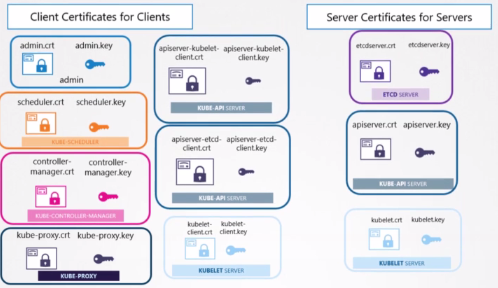
* Note- This (Static password and token file) is not a recommended approach, as it not recommended approach.

**TLS in Kubernetes**

* There are there types of certificates:

1. Server certificates – configured on the servers.
2. Root certificates – configured on the CA servers
3. Client certificates – configured on the client

* Naming convention:
* Usually certificates with public keys are named CRT or pem extension.
* Usually private keys are with extension key or with a dash key in the file names
* The two primary requirements are all the services within the cluster to use server certificates and all clients to use client certificate.
* As we know already, the API server exposes an HTTPS service that other components as well as external users use to manage the Kubernetes cluster. Therefore, it is a server and it requires certificates to secure all communication with its clients.
* We generate a certificate and keep it. We call it apiserver.crt and apiserver.key
* The ETCD server stores all information about the cluster so it requires a pair of certificate and key. We call it etcdserver.crt and etcdserver.key
* The Kubelet servers expose an HTTPS API endpoint that the Kube API server talks to interact with the worker nodes. This required certificate. We call it kubelet.crt and kubelet.key
* The clients who access the kube API server are the administrators through kubectl REST API. The admin user requires a certificate and keep at authenticate the kube API server. We call it admin.crt and admin.key
* The scheduler talks to the Kube API server to look for the pods that require scheduling and then get the API server to schedule the pods on the right worker node. The scheduler is a client that access as kube API server as far as the kube API is concerned the scheduler is just another client like the admin user. So the scheduler needs to validate its identity using a client till a certificate. So it needs its own pair of certificate. We call it scheduler.crt and scheduler.key
* The cube controller manager is another client that accesses the Kube API server. So it also requires a certificate for authentication to the kube API server. We call it controller-manager.crt and controller-manager.key
* The kube-proxy requires a client certificate to authenticate to the Kube-API server. We call it kube-proxy.crt and kube-proxy.key



**TLS in Kubernetes – Certificate Creation**

* To generate the certificate, there are different tools available such as EASYRSA, OPENSSL, CFSSL etc.
* Create a private key using the openness SSL command

#openssl genrsa –out ca.key 2048

* Then, we use the Open SSL requests command along with the key which we created to generate a certificate signing request. The certificate-signing request is like a certificate with all of your details but with no signature. In the certificate-signing request, we specify the name of the component

#openssl req –new –key ca.key -subj “/CN=KUBERNETES-CA” –out ca.csr

* We sign the certificate using the Open SSL x5 0 9 command

#openssl x509 –req –in ca.csr –signkey.ca.key –out ca.crt

* The CA now has its private key and root certificate file.
* Generating the client certificate
* For Admin user
* Create a private key

#openssl genrsa –out admin.key 2048

* We then generate a CSR

##openssl req –new –key ca.key -subj “/CN=KUBERNETES-CA” –out admin.csr

* Signed a certificate using open SSL x509 command

#openssl x509 –req –in admin.csr –CA ca.crt –CAkey ca.key –out admin.crt

* We are signing the certificate with the CA key pair that makes a valid certificate within the cluster. The signed certificate is then output to admin.crt file that is the certificate that the admin user will use to authenticate the Kubernetes cluster.
* If you look at it this whole process of generating a key and certificate pair is similar to creating a user account for a new user. The certificate is the validated user I.D. and the key is like the password.
* We can differentiate between the admin user and normal user by adding the group details for the user in the certificate. We can add group details by adding “/O” parameter while generating a certificate-signing request. Once it is signed, we now have our certificate for the admin user with admin privileges.
* How to use the certificate?

1. By calling an request within the Kubernetes cluster. For example, make an rest api call to the kube-api server.

#curl <https://kube-apiserver:6443/api/v1/pods> --key admin.key –cert admin.crt –cacert ca.crt

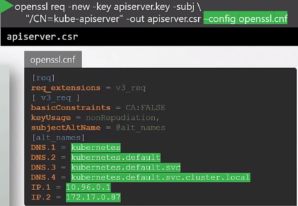
1. The other way of moving is to move all the parameter into a configuration file called “kube.conf” within that, specify api server endpoint details. That is what most of the Kubernetes clients use.

* Note- in Kubernetes, all the carious components to verify each other. They all need a copy of the CA's root certificate. So whenever we configure a server or a client with certificates, we will need to specify the CA root certificate as well.
* We are calling Kube-api server with various name like Kubernetes, Kubernetes.default, kubernetes.default.svc, kubernetes.default.svc.cluster.local and in some place, it is referred to its IP address of the Host. So all of these names must be present in the certificate generated for the Kube API server. Only then, the above name will work and establish the communication. Below is the process to include the entire name and generate the certificate.

1. Generate a key



1. Create a certificate-signing request by mention all the name. For mentioning the entire name, create an open SSL config file. Create an “openssl.cnf” file and specify the alternate names in the “alt\_name” section of the file. Include all the DNS names, the API server goes by as well as the IP address. Then pass this config file as an option while generating the certifacte signing request.

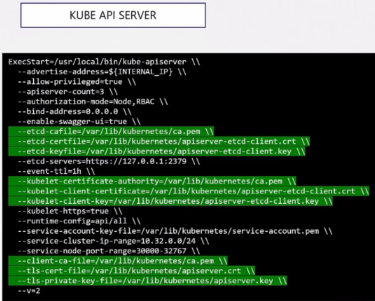


1. Sign the certificate using CA certificate and key. We then have the cube API server certificate.

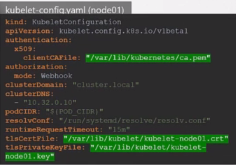


* The location of the certificates are passed into the kube API servers executable or service configuration file

1. CA file needs to be passed in but remember every component CA certificate to verify its client.
2. We provide the API server certificates under the TLS cert option.
3. We then specified the client certificates used by kube-api server to connect to the ETCD server with the CA file.
4. Kube-API client certificate to connect to the Kubelets.



* The Kubelet server is an HTTPS server that runs on each node and responsible for managing the node where the API server talks to.
* To monitor the node as well as send the information regarding what pod to schedule the node.
* We need a key certificate pair for each node in the cluster. The certificate will be named after their nodes like node01, node02 etc. Once the certificates are created then we will use them in the Kubelet config file. We must do it for each node in the cluster



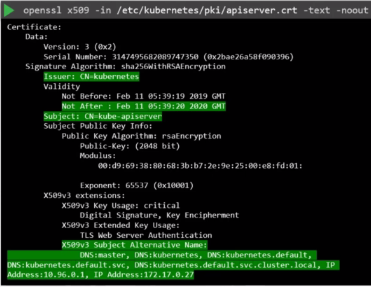
* We know a set of client certificates that will be used by the Kubelet to communicate with the Kube-API server. These are used by the Kubelet to authenticate into the Kube API server. They need to generate as well. Since the nodes are system components, the format starts with the system keyword followed by node and then the node name. For example, node01, node02 etc. the nodes must be added to a group named system nodes.

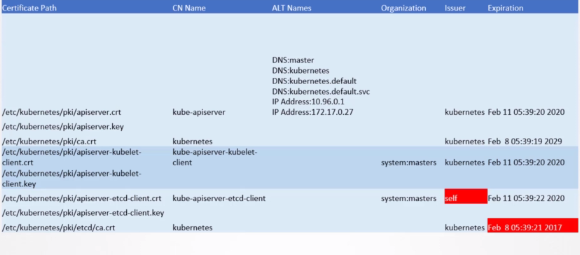
**View Certificate Details**

* If we want to deploy a Kubernetes cluster from scratch, we need to generate the entire certificate.
* We can get the certificate information, if environment is set up by kubeadm, look Kube API server definition file under “/etc/kubernetes/manifests/”
* The command used to start the API server has information about all the certificates it uses. Identify the certificate file used for each purpose and note it down.
* Take each cert and look inside it to find more details about that certificate. For example, API server certificate file. Use the below SSL x509 command to decode the certificate and view the details.

#openssl x509 –in /etc/kubernetes/pki/apiserver.crt –test –noout

* Start with a name on the certificate under the subject section.
* Then check the “Alternative Name”. The Kube-api server has many. So we must ensure all of them are there.
* Then check the validity section of the certificate to identify the expiry date and then the issuer of the certificate.





* We run with an issue, we start looking at logs with the issue then

#journalctl –u etcd.service –l

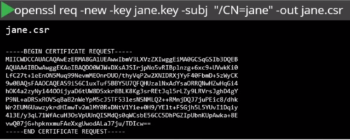
#kubectl logs etcd-master

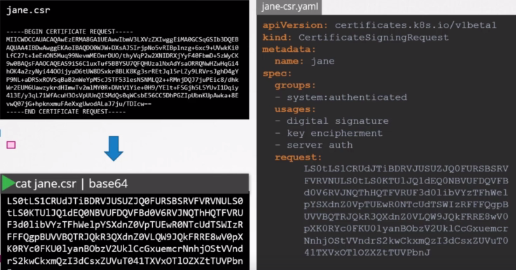
**Certificates API**

* CA is just a pair of key and certificate files that we have generated. Whoever gains access to these file pair of file can sign any certificate for Kubernetes environment. They can create as many user they want.
* In the kerbernetes, the certificate placed in the master node. So the master node is also a RCA server.
* The kubeadm tool does the same thing. It creates a safe pair of files and stores that on the master node itself.
* Kubernetes has a build in certificate API that is helpful in managing the certificate-signing request as well as to rotate certificates when they expire.
* With the certificate API, we send a certificate-signing request directly to Kubernetes through an API call. This time when administrator receives a certificate then he create a Kubernetes API object called certificate-signing request.
* Once the object is created, all certificates signing requests can be seen by administrators of the cluster. The request can be reviewed and approved easily using kubectl commands. this certificate can then be extracted and shared with the user.
* Certificate signing process
* A user creates a key then generates a certificate sending a request using the key with her name on it then sends the request to the administrator



* the administrator takes a key and creates a certificate and request object. The certificate signing request object is created like any other Kubernetes object using a manifest file with the usual fields.





* The request field is where you specify the certificate-signing request sent by the user but we don’t specify as a plain text instead it must be encoded using the base64 command then move the encoded text into the request field and then submit the request once the object is created.
* All certificates sending requests can be seen by administrators by running the “kubectl get csr” command
* Identify the new request and approve the certificate by using below command

#kubectl certificate approve jane

* view the certificate in the yaml format

#kubectl get csr jane –o yaml

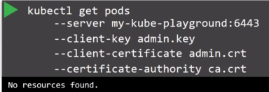
* We will see the certificate in the output but the certifacte is in base64 encoded format. To decode it use the below cammond. This gives the certificate in the plain text format. This can be shared with the user.

#echo ‘LS0..Q0=’ | base64 –decode

* All the certificate related operations are carried out by the controller manager.

**KubeConfig**

* We can specify the key, cert, and cascert in the option in the kubectl.



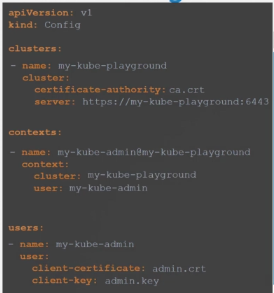
* Typing the “key, cert, and cascert” all the time is tedious, so we move the information to a configuration file called as kubconfig then specify the file in the command



* By default, the kubectl tool looks for a file named config under a directory “.kube” under the user’s home directory. So if we create the config file in this directory then we no need to specifically provide the file path.
* The Kube config file is in a specific format. It has 3 sections.

1. Cluster – this is the various Kubernetes clusters that you need access to.
2. Users - are the user accounts with which you have access to these clusters. These users may have different privileges on different clusters.
3. Contexts – it define which user account will be used to access which cluster.

* Note- we are not configuring any kind of user access or authorization in the cluster with this process we are using existing users with their existing privileges and defining what user we are going to use to access what cluster.
* The server specification in our command goes into the cluster section.
* The admin user’s keys and certificates goes into the user section
* We create a context that specifies to use the user to access the cluster.
* The kube config file is in yaml format. It has three section one for cluster, one for context, one for user. Each of these is in an array format. That way we can specify multiple clusters, Users or contacts within the same file.
* Kubeconfig yaml file



* Note – we don't have to create any object like we usually do for other Kubernetes objects.
* The file is left as it is and is read by the kubectl command and the required value are used.
* WE can specify default context to use by adding a field current context to the kube config file specify the name of the context you use.
* There are command line options available within kubectl to view and modify the cube config files
* To view the current file being used. Run the below command. It list the clusters contacts and users as well as the current context that is set.

#kubectl config view

* If we do not specify which cube config file to use, it ends up using the default file located in the folder “.kube” user’s home directory. Alternatively we can specify a kube config file by passing the cube config option in the command line

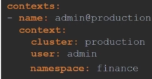
# kubectl config view –kubeconfig=my-custom-config

* How do we change the context to use prod user to access the production cluster?

Below command will change the context

#kubectl config use-context prod-user

* Each cluster may be configured with multiple name spaces within it. We can configure a context to switch to a particular namespace. The context section in the kube config file, can take additional field called namespace where we can specify a particular namespace.



* It’s better to use absolute path in the kubeconfig file when we are providing the certificate path.



**API Groups**

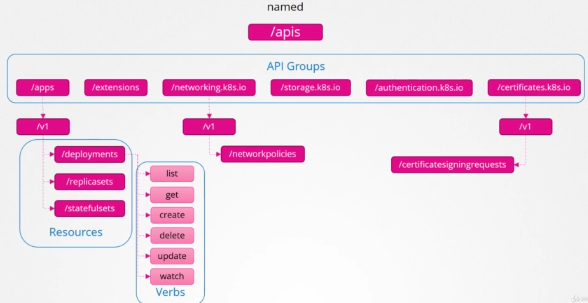
* In this section, we will focus on the API is responsible for the cluster of functionality.
* Below command can show the version of the Kubernetes

#curl https://kube-master:6443/version

* TO get the list of the pods

#curl <https://kube-master:6443/api/v1/pods>

* The Kubernetes API is grouped into multiple such groups based on their purpose such as one for API is one for health one for metrics and logs etc.
* The version API is for viewing the version of the cluster. The matrix and health API are used to monitor the health of the cluster.
* The logs are used for integrating with third party logging applications.
* The API is cagegorise into 2 part:
* The core group - core group is where all core functionality exists such as namespace, pods, replication controllers, events, and points, nodes and bindings, persistent volumes, persistent volume claims, configs, maps, secret, services etc.
* The named group - The named group API’s are more organized and going forward all the newer features are going to be made available to these named groups. It has groups under it for apps, extensions, networking, storage, authentication, authorization etc.



* The Kubernetes API reference page can tell us what the API group is for each object. Select an object and the first section in the documentation page shows its group details.
* List the available API group in the Kubernetes cluster.

#curl <http://localhost:6443> –k

* List all the name api group in the Kubernetes cluster

#curl <http://localhost:6443/apis> -k | grep “name”

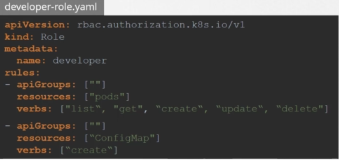
* There is a difference in the Kube proxy and kubectl proxy. Kube proxy is used to enable connectivity between pods and service across different nodes in the cluster. Whereas the kubectl proxy is an HTTPS proxy server created by kubecctl utility to access the kube api server

**Role Based Access Controls**

* We can create a role by using a role object.
* Each rules has 3 section

1. apiGroups – for code group, we can leave API group blank and for any other group you specify the group name
2. resources - the resources that we want to give developers access to our pods.
3. Verbs - The actions that they can take our list get create and delete

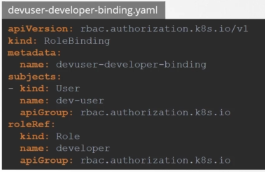
* We can add multiple rule s in the single role
* Role yml file



* We can create a role by below command

#kubectl create –f developer-role.yml

* Link the user to the role. For this we can create another object called role binding. The role binding object links a user object to a role. It has 2 section. The subject is where we specify the user details and the “roleref” section is where we provide the details of the role we created create.



* We can create role binding

#kubectl create –f devuser-developer-binding.yaml

* To view the list of the roles

#kubectl get roles

* To view the role binding

#kubectl get rolebindings

* To view the role in more details

#kubectl describe role <role-name>

* To view the deails of the role binding

#kubectl describe rolebinding <rol-binding-name>

* TO check, if as a user, we have the acess to a particular resource in the cluster

#kubectl auth can-i create deployment ##it will tell if the user has the access to create a deployment in the cluster.

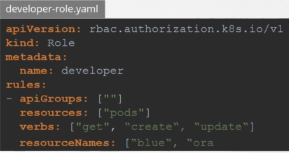
* To test the acess to the create user

#kubectl auth can-I create deployment –as dev-user

* To test if a user has the access to a particular resource in a particular namespace

#kubect auth can-I create pods –as dev-user –namespace test

* We can restrict access based on then number of accessing pod.



**Cluster Roles and Role Bindings**

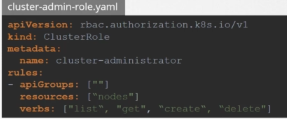
* When we said that roles and role bindings are named spaced meaning they are created within name spaces.
* The resources are categorized as either namespace or clusters Scoped.
* The namespace resource like pods, replica set, jobs, deployment, service, secret etc. They created in the snamespace. We specify a name space when we create them. If we don’t specify the namespace then it will create in the default namespace. To see the full list of the name space resource, run the below command.

#kubectl api-resources –namespaced=true

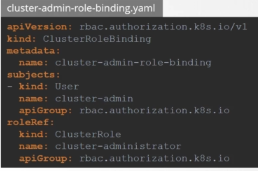
* The cluster Scoped resources are those where we don't specify a namespace when we create them like node, persistent volume, persistent cluster roles, certificate signing request, name space object. To view the full list of the cluster scoped resource.

#kubectl api-resource –namespace=false

* Clusterroles are just like roles except they are for cluster scoped resources. For example a cluster admin role can be created to provide a cluster administrator permissions to view create or delete nodes in a cluster. Similarly if storage administrator role can be created to authorize a storage admin to create persistent volumes and claims.
* Cluster role yaml file



* Link the user to the cluster role. For this we create another object. Below is the Yaml file.



* We can create role binding

#kubectl create –f devuser-admin-role-binding.yaml

* Note - cluster rules and bindings are used for clusters of resources but that is not a hard rule we can create a cluster rule for namespace resources as well. When you do that the user will have access to these resources across all name spaces.

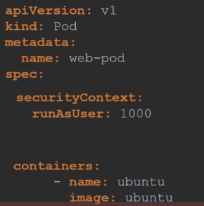
**Image Security**

* The image name in the Kubernetes follows docker image naming conventions. For example, when we say “ngnix” image. So its actually means image repository. So inour case it will actually be look like “ngnix/ngnix” which means “user-account/Image-repository”
* By default, the Kubernees pull the image from the docker repository.
* To use an image from the private repository, we replace the image name with the full path to the one in the private registry. We can implement the authentication to the private repository. So for this, we can create secret object with credential. Docker registry is a built in secret type that was built for storing Docker credentials. Then specify the secret in the pod definition file.



**Security Contexts**

* In Kubernetes, containers are encapsulated in pods. We may choose to configure the security settings at a container level or at a pod level.
* If we configure the security at a pod level, the settings will carry over to all the containers within the pod.
* If we can figure security at both the pod and the container then the settings on the container will override the settings on the pod.
* Pod definition based pod level security



* Pod definition based on the container level security



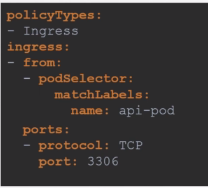
* Capabilities are only supported at the container level and not at the POD level.

**Network Policy**

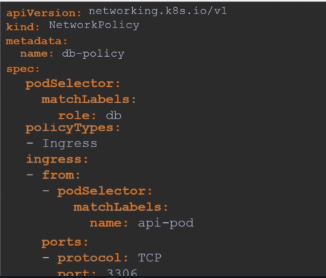
* There are basically two type of traffic:

1. Ingress – incoming traffic from the user is called ingress
2. Egress – outgoing request from the app server is called egress.

* One of the prerequisites for networking in Kubernetes networks is whatever solution you implement the pods should be able to communicate with each other without having to configure any additional settings like routes.
* Kubernetes is configured by default with an all allow rule that allows traffic from any pod to any other pod or services within the cluster.
* Network policy is another object in the Kubernetes namespace just like pods, replica set, or services.
* We link a network policy to one or more pod. We can define rules within the network policy.
* How do you apply or link a network policy to a pod?
* We will use labels and selector to link pod and network policy.
* We labeled the pod and used the same labels on the pod selector field in the network policy and then we build our rule.
* If we want to allow only ingress traffic from the DB host



* Then we specify the ingress rule that allows traffic from the API pod



* Run the below command to create the policy

#kubectl create –f policy-defination.yaml

* Note - Network policies are enforced by the network solution implemented on Kubernetes cluster and not all network solutions support network policies. Below are the few of them that are supported are:

Kube-router

Calico

Ramana

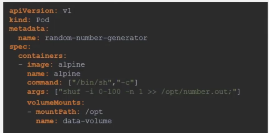
Weave-net

* “Flannel” network solution do not support network ploicies.

**Storage**

**Volumes**

* We can configure the volumes in Kubernetes in different ways. For example, we will create a directory in a host. This is fine for single node cluster but it is not recommended for a production environment. Below entry we will make in the pod definition file.



* Here, “volumeMounts” is mounted inside the container.
* Below will create a volumes in the kuberetes.



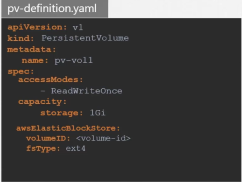
* Kubernetes support various times of the storage solution such as NFS, GlusterFS, FLocker, Fiber channel, SCALEIO or public cloud solution like AWS EBS, Azure Disk or file, Google persistent disk.
* For example to configure and AWS elastic block store volume as the storage option for the volume. We will use the below configuration.



Where, “ftype” is the file system type.

**Persistent Volumes**

* A persistent volume is a cluster wide pool of storage volumes configured by an administrator to be used by users deploying applications on the Cluster. The users can now select storage from this pool using persistent volume claims.
* We can create persistent Volume by below configuration.



* Under the specs section specify the access modes access
* access more defines how a volume should be mounted on the hosts whether in a read only mode or read write mode etc.

The supported values are:

1. ReadOnlyMany
2. ReadWriteOnce
3. ReadWriteMany

* “capacity” specify the amount of storage to be reserved for the persistent volume.
* “hostpath” uses storage from the nodes local directory. Note- this option is not to be used in a production environment.
* To create a volume use the below command.

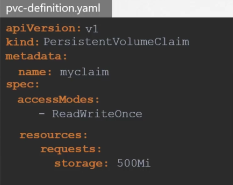
#kubectl create –f pv-defination.yml

* TO view the persistent volumes

#kubectl get persistentvolume

**Persistent Volume Claims**

* + Persistent volumes and persistent volume claims are two separate objects in the Kubernetes namespace.
  + An administrator creates a set of persistent volumes and the user creates persistent volume claims use to storage.
  + Once the persistent volume claims are created, Kubernetes binds the persistent volumes to claims based on the request and properties set on the volume
  + Every persistent volume claim is bound to a single persistent volume.
  + During the binding process, Kubernetes tries to find a persistent volume that has sufficient capacity as requested by the claim and any other request properties such as access modes volume modes storage class etc.
  + If there are multiple possible matches for a single claim and we would like to specifically use a particular volume, we could still use labels and selectors to bind to the right volumes.
  + Note - A smaller claim may get bound to a larger volume if all the criteria matches and there are no better options.
  + There is a one to one relationship between claims and volumes. so no other claims can utilize the remaining capacity in the volume.
  + If there are no volumes available the persistent volume claim will remain in a pending state until newer volumes are made available to the cluster. Once newer volumes are available. The claim would automatically be bound to the newly available volume.
  + We can create a persistent volume claim by using below definition file



Run the below command to create persistent volume claim.

#kubectl create –f pvc-defination.yml

To view the persistent volume claim

#kubectl get persistentvolumeclaim

* To delete a persistent volume claim, we can use below command

#kubectl delete persistentvolumeclaim myclaim

* The persistent volume calm is set to “persistentVolumeReclaimPolicy: Ratain” option by default which means, the persistent volume will remain until it is manually deleted by the administrator. It is not available for reuse by any other claims.
* “persistentVolumeReclaimPolicy: Delete” will delete automatically which means once the claim is deleted then volume will be delete automatically.
* “persistentVolumeReclaimPolicy: Recycle” , the data in the data volume will be scrubbed before making it available to other claims.

**Networking**

**Prerequisite - Switching Routing**

* + Ip link - to see the interface on the server
  + Route - It displays the kernels routing table
  + Assigning the IP address to the internfate

#ip addr 192.168.1.10/24 dev eth0

* + Route – display the kernel routing table
  + Add the route to the table. Add a network 192.168.2.0/24 via 192.168.1.1

#ip route add 192.168.2.0/24 via 192.168.1.1

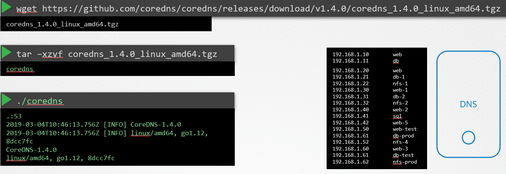
#ip route add default via 192.168.2.1 ##adding the default routing table

**Prerequisite – DNS**

* + DNS resolution file : /etc/resolve.conf
  + A –record : Stores the IP address of the host
  + AAAA – record : Stores the IPv6 IP address of the host
  + CNAME – Mapping one name to another name

**Prerequisite – CoreDNS**

* Installing the code DNS



* + - by default, it listens on port 53.
    - We can specify the Ip address in the “/etc/hosts” file
  + Read more about the CoreDNS

<https://github.com/kubernetes/dns/blob/master/docs/specification.md>

<https://coredns.io/plugins/kubernetes/>

**Prerequisite – Network Namespaces**

* + Network namespaces are used by containers like Docker to implement network isolation.
  + Containers are separated from the underlying host using namespaces.
  + When container is created then we create a network name space for it in a way that it has no visibility to any network related information on the host.
  + Within its namespace, the container can have its own virtual interfaces, routing and ARP tables. The container has its own interface.
* To create a new network namespace on a Linux host run the below command.

#ip netns add red

* To list the interfaces, run the below command

#ip netns

* Run the command inside the name space

# ip netns exec <name-space> ip link

* Stabling the connectivity between the name space
* To create a virtual cable

#ip link add <virtual-eth1> type veth peer name <virtual-eath2>

* Attach each interface to the appropriate namespace.

#ip link set <virtual-eth1> netns <name-space1>

#ip link set <virtual-eth2> netns <name-space2>

* Assign the IP address to each of the namespace.

#ip –n <name-space1> addr add 192.168.15.1 dev <virtual-eth1>

#ip –n <name-space2> addr add 192.168.15.2 dev <virtual-eth2>

* Bring up the interface

# ip –n <name-space1> link set <virtual-eth1> up

# ip –n <name-space2> link set <virtual-eth2> up

* Run the below command to check the connectivity

# ip netns exec <name-space1> ping 192.168.15.2

#ip netns exec <name-space1> arp

#ip netns exec <name-space2> arp

* How to connect multiple name space
* Create a switch. Here we will create a virtual bridge on the host system

# ip link add v-net-0 type bridge

#ip link set dev v-net-0 up

#ip –n <name-space1> link del <virtual-eth1> ##this will delete the link between two name space.

#ip link add veth-red type veth peer < v-net-0> ##this will create a pipe between the name space and the switch

# ip link add veth-blue type veth peer name veth-blue-br ## this will create a pipe between the name space and the switch

#ip link set veth-red netns red

#ip link set veth-red-br master v-net ##this will attach the name space to the network

# ip –n red addr add 192.168.15.1 dev veth-red ##set the IP address of the to the name space

# ip –n blue addr add 192.168.15.2 dev veth-blue ##set the IP address of the to the name space

#ip –n red link set veth-red up ##to start the link

#ip –n blue link set veth-blue up ##to start the link

Connect the name spaces between them

**Prerequisite - Docker Networking**

Prerequisite – CNI (Container Network Interface)

**Cluster Networking**

* + K8s-master accepts the connection on 6443 for the API server
  + The kubelet on the master and worker node listen on port 10250.
  + The kube-scheduler required listen on port 10251.
  + Kube-controller-manager listen on port 10252
  + The worker node exposes services for external access on port 30000 to 32767.
  + ETCD server listens on port 2379.
  + ETCD clients can communicate with each other at port 2380.

**Pod Networking**

* + Every POD should have an IP address.
  + Every POD should be able to communicate with every other POD in the same node.
  + Every POD should be able to communicate with every other POD on other nodes without NAT.

**CNI in Kubernetes**

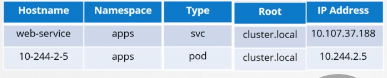
**CNI weave**

**IP Address Management – Weave**

**Service Networking**

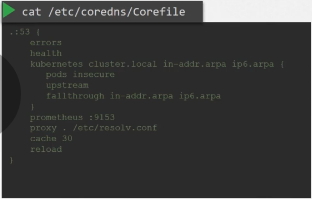
**DNS in Kubernetes**

* + When service is created in K8s then all the service is grouped under sub domain “<name-space>” and “<name-space> is grouped under “svc” and “svc” is grouped under “cluster.local”. So we can access the service under k8s is **“<service-name>.“<name-space>.svc. cluster.local**”
  + For Pods, the DNS record is not updated by default But we can add the recoded them explicitly.
  + For each Pod, K8s replace IP address with a name by replacing “.” To “-“ . Then the name is grouped under sub domain “<name-space>” and “<name-space> is grouped under “pod” and “pod” is grouped under “cluster.local”. So we can access the pod under k8s is **“<pod-name>.“<name-space>.pod. cluster.local**”



**CoreDNS in Kubernetes**

* + Prior to k8s version v1.12, K8s uses “kube-dns” as a DNS server.
  + After the k8s version v1.12, it is recommended to use “CoreDNS” as DNS server.
  + CoreDNS setps:
* Configuration file:



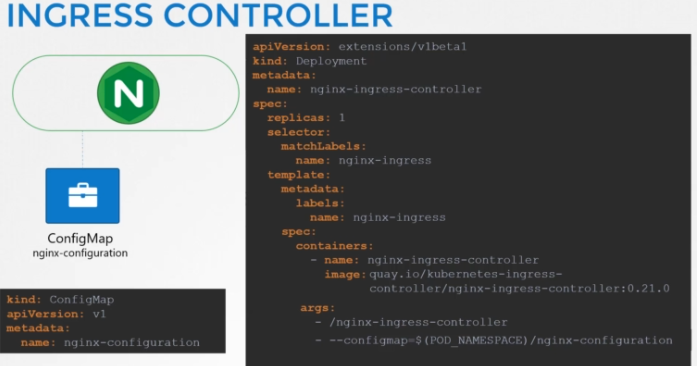
**Ingress**

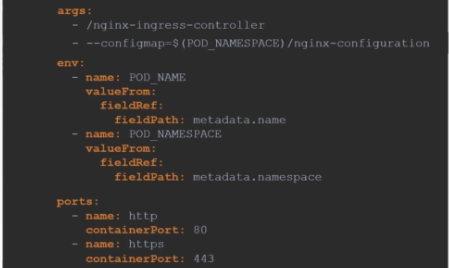
* + We can say, it is layer 7 load balancer in the K8s.
  + How to configure Ingress?
* We can configure in 2 steps:

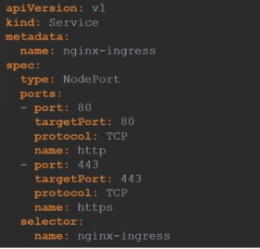
1. Deploy a ingress controller
2. The configure the set of rules and the resource

* An Ingress Controller does not come with K8s by default.

* Ingress Controller -
* There are number of ingress controller available which are HTTP Load Balancer, NGINX, Contour, HAPROXY, TRAFIK and Istio. Out of this, **GCE and NGINX** are currently being supported and maintained by the Kubernetes project.
* Deploy the NGINX in the k8s.

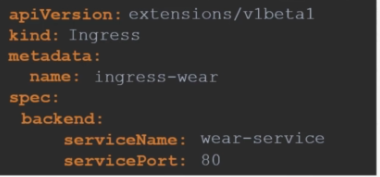




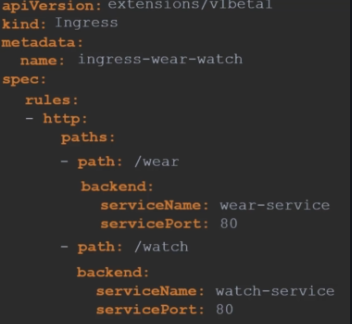




* Ingress Resources
  + - It is a set of rules and configurations applied on the ingress controller.
    - Ingress setup
* This is for single service.



* This is for multiple rules based on the condition



* The setup for multiple hostname /domain name

